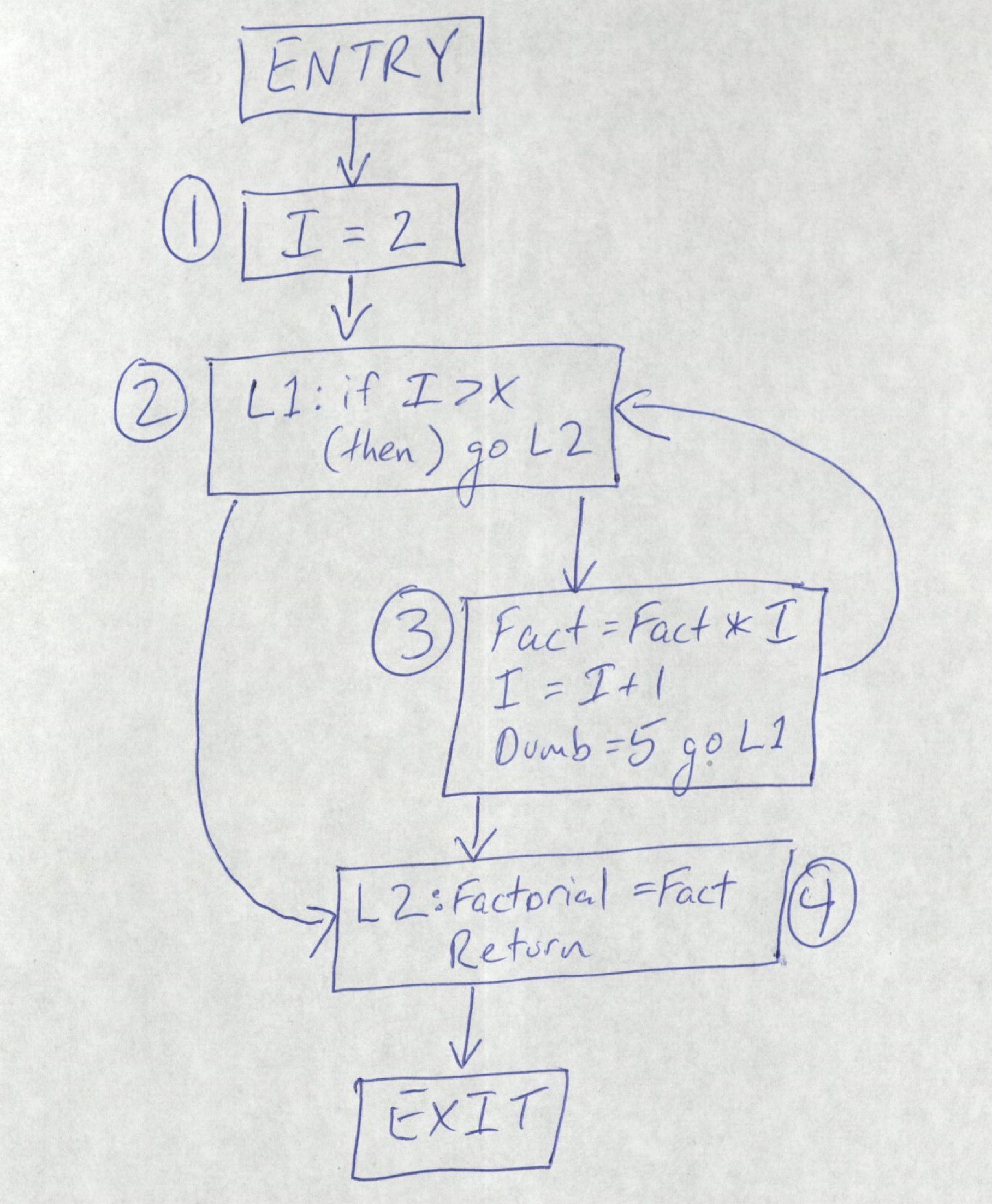
Alexander Powell

09.21.2016

CSCi 680 – Compiler Optimization HW #1

1. Control Flow:
   1. Based on the line numbers of the given code, the set of basic blocks can be written as:

{ {1}, {2, 3}, {4, 5, 6}, {7, 8} }



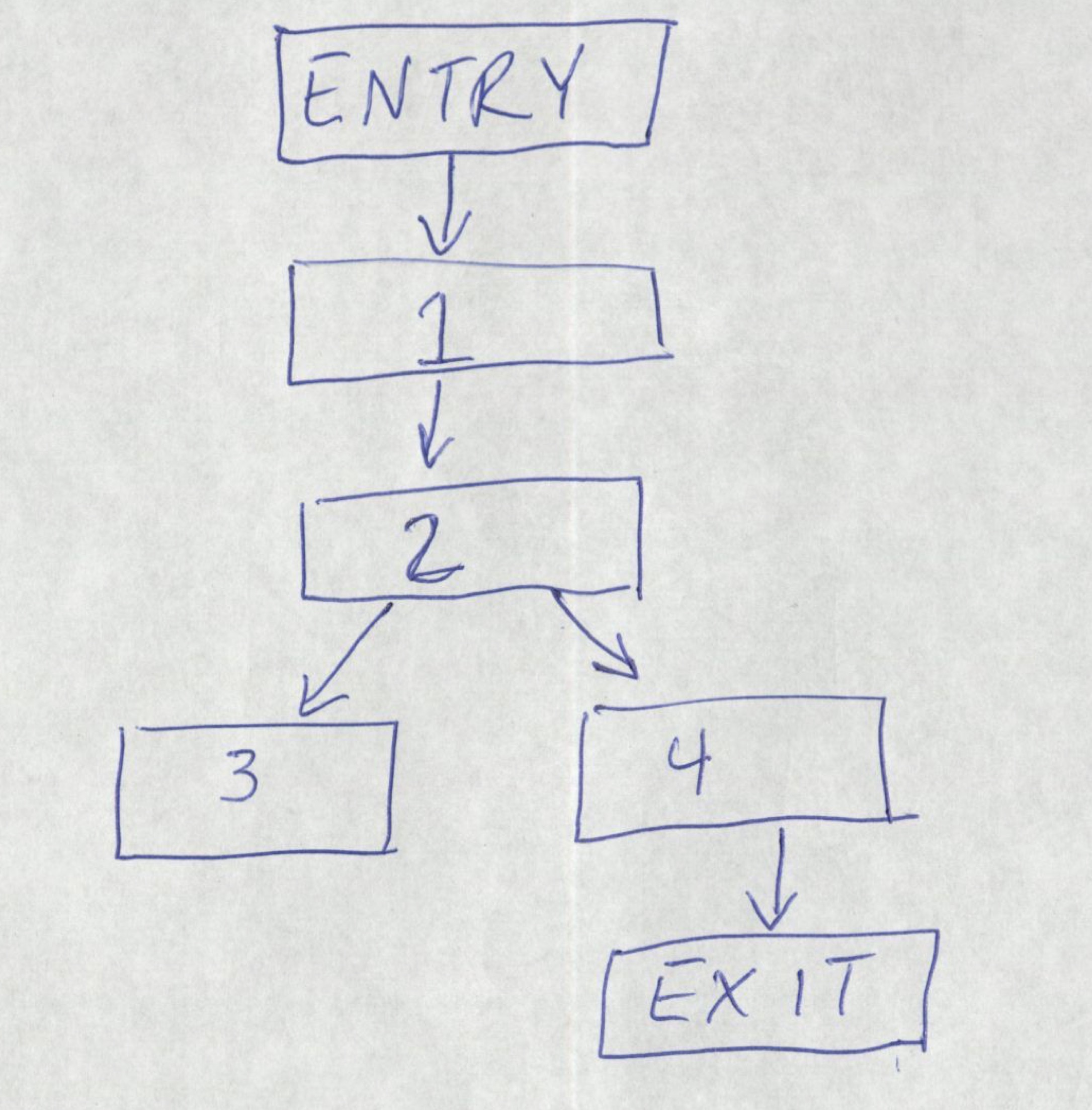
* 1. The immediate dominators can be described by the following relationship:

1 🡪 ENTRY 2 🡪 1

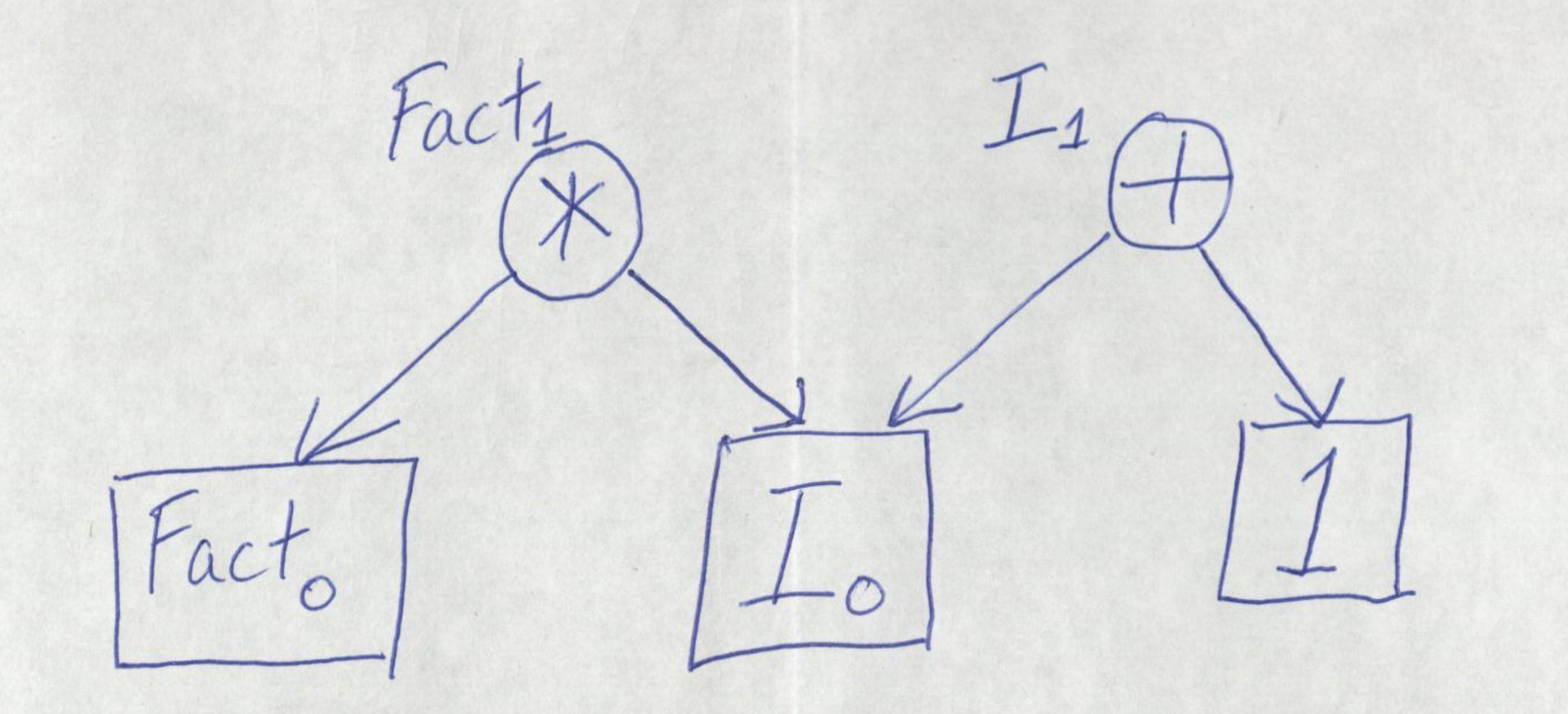
3 🡪 2 4 🡪 2

EXIT 🡪 4

Therefore, the dominator tree appears as:



* 1. The basic blocks 2 and 3 form a cycle. To prove that this cycle is a loop we must show it is a strongly connected component (SCC) with a single entry point. Clearly {2, 3} form an SCC because each node in the set is reachable from every other node in the set. That is, 2 is reachable from 3, and 3 is reachable from 2. Also, there’s clearly only one entry point so the cycle is a loop.
  2. The following is a DAG (directed acyclic graph) of the largest basic block. Note that the variable dumb is omitted.



1. Data Flow Analysis:
   1. The domain for this three address code can be described as the power-set of all the variables defined. The code has a forward direction. The initializations for each basic block are at lines 1, 5, 8, 9, and 11.

GEN[n] is the set of all variables that are read by n, and KILL[n] is a singleton set containing the variable that is written by n.

Then transfer functions IN and OUT for each basic block are shown below.

OUT[BEGIN] = {m, n, u1, u2, u3}

IN[BB1] = {m, n, u1, u2, u3}

OUT[BB1] = {i, j, u2, u3}

IN[BB2] = {i, j, u2, u3}

OUT[BB2] = {u2, u3, j}

IN[BB3] = {u2, u3, j}

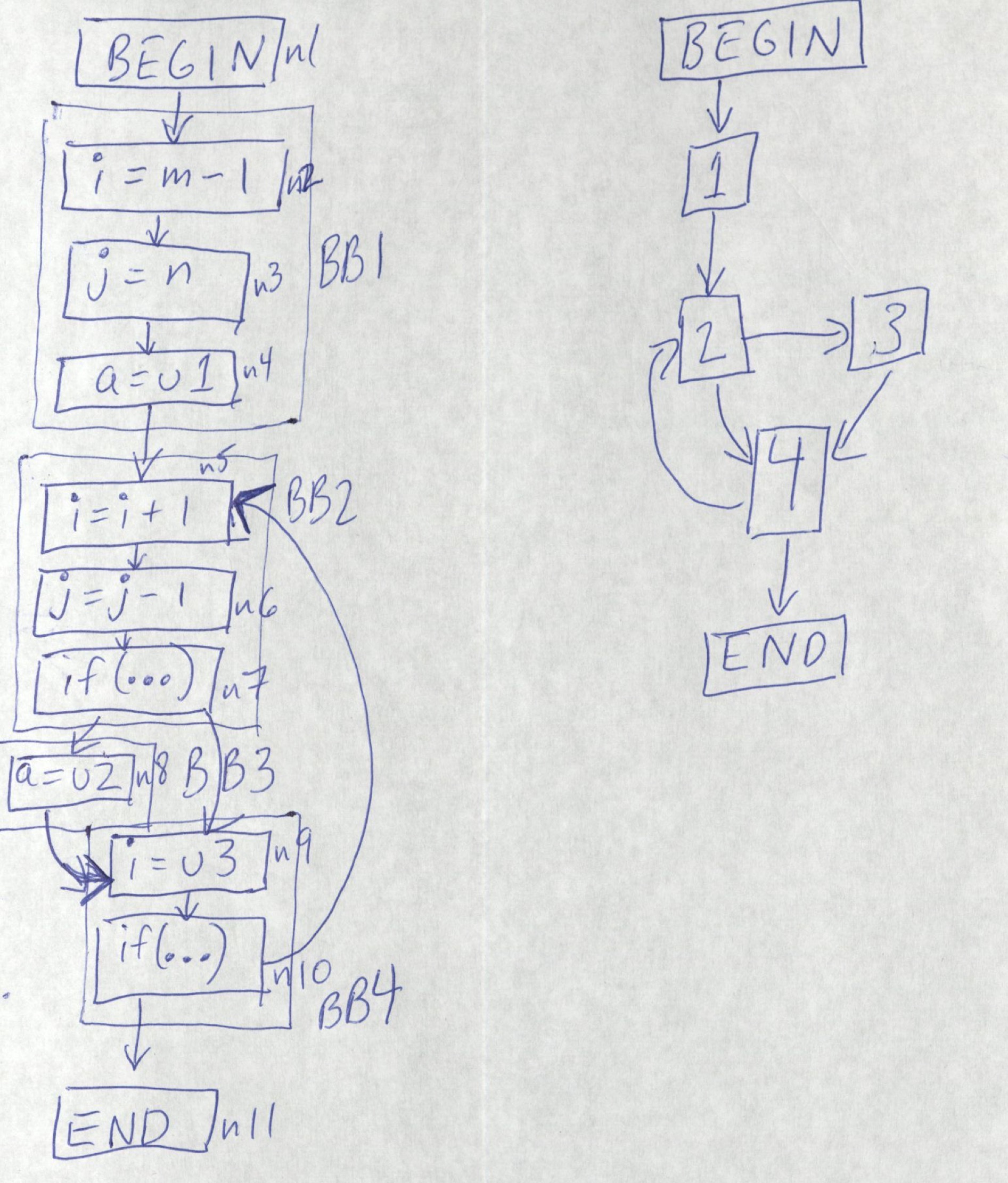
OUT[BB3] = {u3, j, u2}

IN[BB4] = {u3, j, u2}

OUT[BB4] = {i, j, u2, u3}

IN[END] = { }

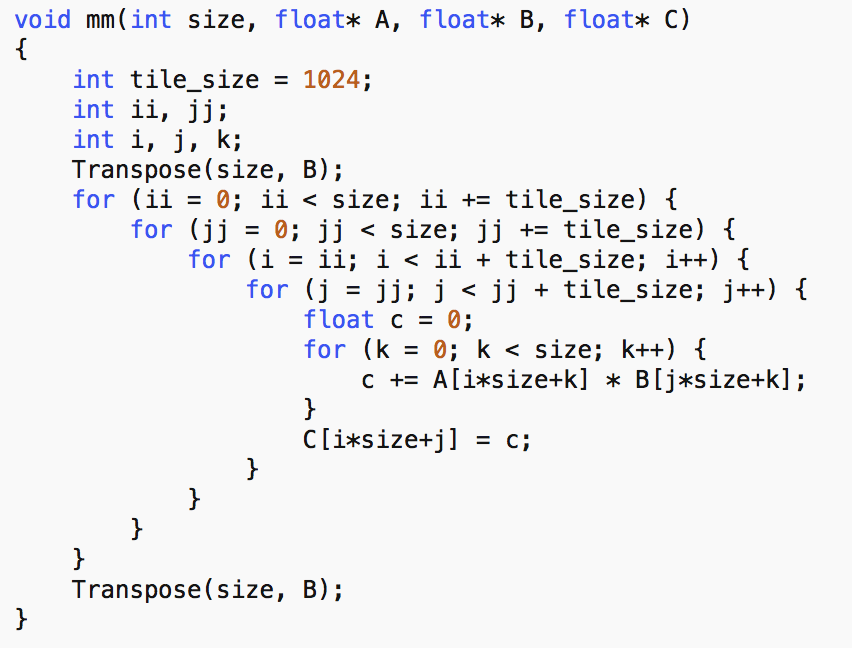
* 1. The control flow graph is shown below. The left one shows the basic blocks with their lines inside, and on the right is a condensed version showing just the basic blocks with entry and exit nodes.



* 1. The live variable analysis is shown in the table below

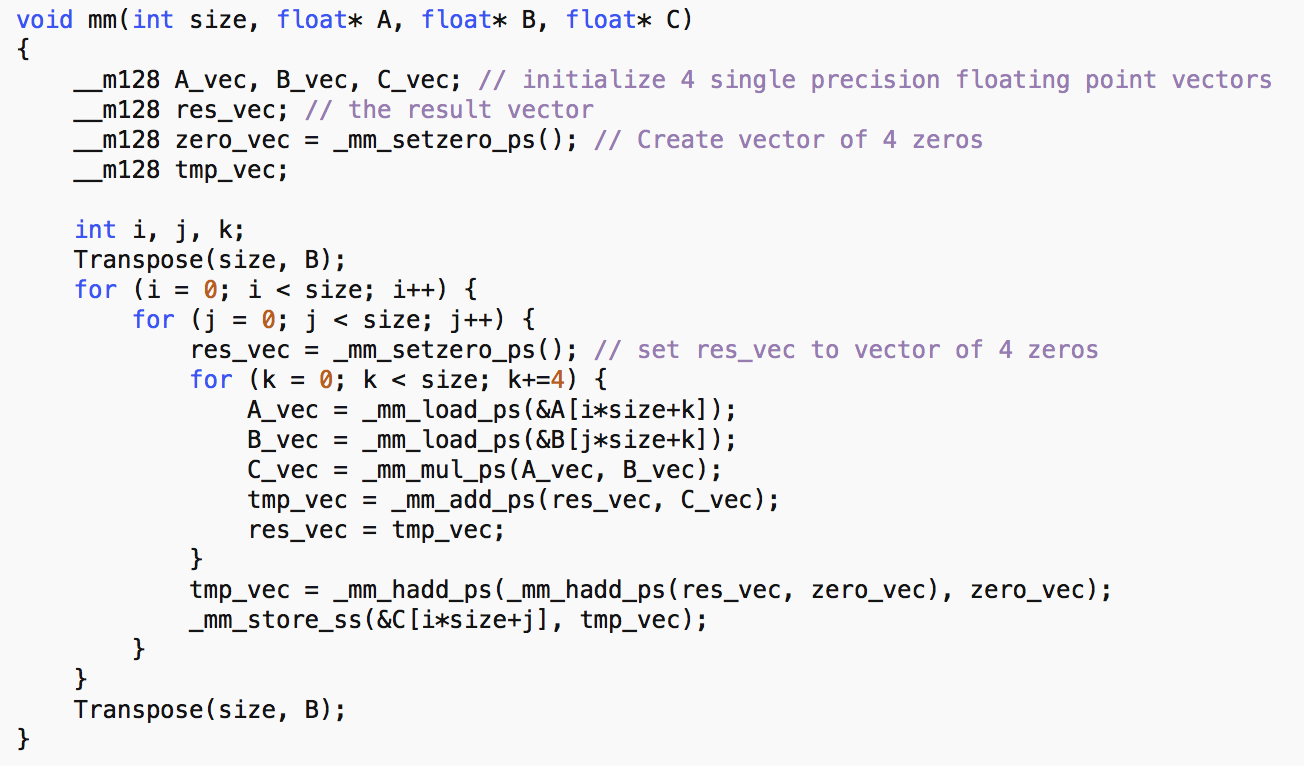
|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Iteration\BB |  | BB1 | BB2 | BB3 | BB4 |
| 0 | IN | m, n, u1, u2, u3 | j, u2, u3 | u2, u3 | u3, j, u2 |
| 0 | OUT | i, j, u2, u3 | j, u3 | u3, j, u2 | i, j, u2, u3 |
| 1 | IN | m, n, u1, u2, u3 | j, u2, u3 | u2, u3, j | u3, j, u2 |
| 1 | OUT | i, j, u2, u3 | j, u3 | u3, j, u2 | i, j, u2, u3 |
| 2 | IN | m, n, u1, u2, u3 | i, j, u2, u3 | u2, u3, j, i | u3, j, u2 |
| 2 | OUT | i, j, u2, u3 | u2, u3, j | u3, j, u2 | i, j, u2, u3 |

1. Matrix Multiplication Optimization:
   1. CPU Configurations:
      1. MacBook Pro
      2. Processor: 2.7 GHz Intel Core i5
      3. Memory: 8 GB 1867 MHz DDR3
      4. 256 GB Flash Storage
      5. L1 Cache Size: 32768
      6. L2 Cache Size: 262144
      7. L3 Cache Size: 3145728
      8. Compiler Type: GCC
      9. Compiler Optimization Level: 03
         * Compiled with “-O3 -msse4.1”
   2. The code for the tiling section of the matrix multiplication is shown below:



The following plot shows the execution times based on different tile sizes. As you can see, there wasn’t much difference in execution time between the different tile sizes except when really large tile sizes were used (512 and 1024).

* 1. The code for the vectorized kernel loops looks like:



* 1. The graph below shows the difference in performance between the base version, transposed version, best tiled version, and the vectorized version.

Here we can clearly see that the base version takes much longer to perform the calculation than even the transposed version. The transposed version alone takes only about one tenth the time of the base. The best of the tiled versions, which was with a tile size of 128 was just a bit faster than the transpose version. However, we saw considerable improvements in the vectorized version, which took only about a third of the time as the tiled and transposed versions. The reason that the vectorized version is so fast is because by using the SSE intrinsic functions, we are performing operations in parallel on arrays on 4 floating point values.